PHASE1-1.1 PART 1: Cost Agent Skeleton - Code Generation

OptiInfra Development Series
Phase: Cost Agent (Week 2-3)
Component: Cost Agent Foundation
Estimated Time: Already Complete (from P-03)
Dependencies: P-01 (Bootstrap), P-03 (Cost Agent Skeleton), 0.10 (Shared Utilities)

📋 Overview

This document covers the Cost Agent Skeleton that was already created in **P-03 (PILOT Phase)**. Since this component is already complete, this guide focuses on:

- Reviewing what was built
- Verifying it's ready for Phase 1 development
- Ensuring integration points are correct
- Validating the foundation before adding collectors

🎯 Status Check

<div style="background: #e8f5e9; border-left: 4px solid #00C853; padding: 15px; margin: 15px 0;">

✅ ALREADY COMPLETE (from P-03)

The Cost Agent Skeleton was created during the PILOT phase and includes:

- FastAPI application structure
- Health check endpoint
- Database connections
- Logging configuration
- Basic metrics endpoint
- Testing framework

</div>

🔍 What Was Built in P-03

1. Project Structure

```
services/cost-agent/
├── src/
│   ├── __init__.py
│   ├── main.py              # FastAPI application
│   ├── config.py            # Configuration management
│   ├── api/
│   │   ├── __init__.py
│   │   ├── health.py        # Health check endpoints
│   │   └── analyze.py       # Analysis endpoints (from P-04)
│   ├── models/
│   │   ├── __init__.py
│   │   └── analysis.py      # Data models
│   ├── workflows/
│   │   ├── __init__.py
│   │   ├── state.py         # LangGraph state
│   │   └── cost_optimization.py  # Base workflow
│   ├── nodes/
│   │   ├── __init__.py
│   │   ├── analyze.py       # Analysis node
│   │   ├── recommend.py     # Recommendation node
│   │   └── summarize.py     # Summary node
│   ├── utils/
│   │   └── __init__.py
│   └── metrics.py           # Prometheus metrics (from 0.11)
├── tests/
│   ├── __init__.py
│   ├── test_health.py
│   ├── test_workflow.py
│   └── test_analyze_api.py
├── requirements.txt
├── Dockerfile
└── README.md
```

2. Core Components

Main Application (src/main.py)

python

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from prometheus_client import make_asgi_app
import logging

from .api import health, analyze
from .config import settings
from shared.utils.database import get_postgres_connection
from shared.utils.prometheus_metrics import FastAPIMetricsMiddleware

# Initialize FastAPI app
app = FastAPI(
    title="OptiInfra Cost Agent",
    description="AI-powered cost optimization agent",
    version="1.0.0"
)

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Add Prometheus metrics middleware
app.add_middleware(FastAPIMetricsMiddleware, app_name="cost-agent")

# Include routers
app.include_router(health.router, prefix="/api/v1", tags=["health"])
app.include_router(analyze.router, prefix="/api/v1", tags=["analysis"])

# Metrics endpoint
metrics_app = make_asgi_app()
app.mount("/metrics", metrics_app)

@app.on_event("startup")
async def startup_event():
    """Initialize connections on startup."""
    logger = logging.getLogger(__name__)
    logger.info("Starting Cost Agent...")

    # Test database connection
    try:
        conn = get_postgres_connection()
        conn.close()
        logger.info("Database connection successful")
    except Exception as e:
        logger.error(f"Database connection failed: {e}")
        raise

@app.on_event("shutdown")
async def shutdown_event():
    """Cleanup on shutdown."""
    logger = logging.getLogger(__name__)
    logger.info("Shutting down Cost Agent...")

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(
        "src.main:app",
        host="0.0.0.0",
        port=8001,
        reload=True,
        log_level="info"
    )
```

Configuration (src/config.py)

python

```python
from pydantic_settings import BaseSettings
from typing import Optional

class Settings(BaseSettings):
    """Application settings."""

    # Service
    SERVICE_NAME: str = "cost-agent"
    SERVICE_PORT: int = 8001
    DEBUG: bool = False

    # Database
    DATABASE_URL: str = "postgresql://optiinfra:password@localhost:5432/optiinfra"
    CLICKHOUSE_URL: str = "http://localhost:8123"
    QDRANT_URL: str = "http://localhost:6333"
    REDIS_URL: str = "redis://localhost:6379"

    # Orchestrator
    ORCHESTRATOR_URL: str = "http://localhost:8080"

    # LLM (will be used in 1.8)
    OPENAI_API_KEY: Optional[str] = None
    ANTHROPIC_API_KEY: Optional[str] = None

    # Cloud Providers (will be used in 1.2-1.4)
    AWS_ACCESS_KEY_ID: Optional[str] = None
    AWS_SECRET_ACCESS_KEY: Optional[str] = None
    AWS_REGION: str = "us-east-1"

    GCP_PROJECT_ID: Optional[str] = None
    GCP_CREDENTIALS_PATH: Optional[str] = None

    AZURE_SUBSCRIPTION_ID: Optional[str] = None
    AZURE_TENANT_ID: Optional[str] = None
    AZURE_CLIENT_ID: Optional[str] = None
    AZURE_CLIENT_SECRET: Optional[str] = None

    # Analysis
    ANALYSIS_LOOKBACK_DAYS: int = 30
    SPOT_SAVINGS_TARGET: float = 0.35  # 35% target
    RI_SAVINGS_TARGET: float = 0.50    # 50% target
    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"

settings = Settings()
```

Health Check API (src/api/health.py)

python

```python
from fastapi import APIRouter, HTTPException
from pydantic import BaseModel
from datetime import datetime
import logging

from shared.utils.database import (
    get_postgres_connection,
    get_clickhouse_connection,
    get_qdrant_client,
    get_redis_connection
)

router = APIRouter()
logger = logging.getLogger(__name__)

class HealthResponse(BaseModel):
    """Health check response."""
    status: str
    timestamp: datetime
    version: str
    database: dict

@router.get("/health", response_model=HealthResponse)
async def health_check():
    """Health check endpoint."""

    database_status = {
        "postgres": "unknown",
        "clickhouse": "unknown",
        "qdrant": "unknown",
        "redis": "unknown"
    }

    # Check PostgreSQL
    try:
        conn = get_postgres_connection()
        conn.close()
        database_status["postgres"] = "healthy"
    except Exception as e:
        logger.error(f"PostgreSQL health check failed: {e}")
        database_status["postgres"] = "unhealthy"
    # Check ClickHouse
    try:
        client = get_clickhouse_connection()
        client.command("SELECT 1")
        database_status["clickhouse"] = "healthy"
    except Exception as e:
        logger.error(f"ClickHouse health check failed: {e}")
        database_status["clickhouse"] = "unhealthy"

    # Check Qdrant
    try:
        client = get_qdrant_client()
        client.get_collections()
        database_status["qdrant"] = "healthy"
    except Exception as e:
        logger.error(f"Qdrant health check failed: {e}")
        database_status["qdrant"] = "unhealthy"

    # Check Redis
    try:
        redis = get_redis_connection()
        redis.ping()
        database_status["redis"] = "healthy"
    except Exception as e:
        logger.error(f"Redis health check failed: {e}")
        database_status["redis"] = "unhealthy"

    # Overall status
    all_healthy = all(status == "healthy" for status in database_status.values())
    overall_status = "healthy" if all_healthy else "degraded"

    return HealthResponse(
        status=overall_status,
        timestamp=datetime.utcnow(),
        version="1.0.0",
        database=database_status
    )

@router.get("/ready")
async def readiness_check():
    """Kubernetes readiness probe."""
    try:
        conn = get_postgres_connection()
        conn.close()
        return {"status": "ready"}
    except Exception as e:
        logger.error(f"Readiness check failed: {e}")
        raise HTTPException(status_code=503, detail="Service not ready")

@router.get("/live")
async def liveness_check():
    """Kubernetes liveness probe."""
    return {"status": "alive"}
```

Cost Agent Metrics (src/metrics.py)

python

```python
from prometheus_client import Counter, Gauge, Histogram
from shared.utils.prometheus_metrics import BaseMetrics

class CostAgentMetrics(BaseMetrics):
    """Cost Agent specific metrics."""

    def __init__(self):
        super().__init__(service_name="cost-agent")

        # Cost savings metrics
        self.cost_savings_total = Counter(
            'cost_savings_total',
            'Total cost savings in USD',
            ['provider', 'optimization_type']
        )

        self.cost_recommendations_total = Counter(
            'cost_recommendations_total',
            'Total number of cost recommendations',
            ['type', 'confidence']
        )

        # Spot instance metrics
        self.spot_migration_success_rate = Gauge(
            'spot_migration_success_rate',
            'Success rate of spot instance migrations'
        )

        self.spot_migration_attempts = Counter(
            'spot_migration_attempts_total',
            'Total spot migration attempts',
            ['outcome']
        )

        # Reserved instance metrics
        self.reserved_instance_coverage = Gauge(
            'reserved_instance_coverage',
            'Percentage of instances covered by RIs',
            ['provider']
        )
        self.reserved_instance_recommendations = Counter(
            'reserved_instance_recommendations_total',
            'RI recommendations generated',
            ['provider', 'instance_type']
        )

        # Right-sizing metrics
        self.rightsizing_opportunities = Gauge(
            'rightsizing_opportunities',
            'Number of right-sizing opportunities identified',
            ['provider']
        )

        self.rightsizing_savings_potential = Gauge(
            'rightsizing_savings_potential_usd',
            'Potential savings from right-sizing in USD',
            ['provider']
        )

        # Analysis metrics
        self.analysis_duration_seconds = Histogram(
            'cost_analysis_duration_seconds',
            'Time spent analyzing costs',
            ['provider'],
            buckets=[1, 5, 10, 30, 60, 120, 300]
        )

        self.idle_resources_detected = Counter(
            'idle_resources_detected_total',
            'Number of idle resources detected',
            ['provider', 'resource_type']
        )

# Global metrics instance
cost_metrics = CostAgentMetrics()
```

3. Dependencies (requirements.txt)

txt

```
# Web Framework
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
pydantic-settings==2.1.0

# LangGraph (from P-04)
langgraph==0.0.25
langchain==0.1.0
langchain-core==0.1.0

# Database Clients
psycopg2-binary==2.9.9
clickhouse-driver==0.2.6
qdrant-client==1.7.0
redis==5.0.1

# Metrics
prometheus-client==0.19.0

# Cloud SDKs (will be used in 1.2-1.4)
boto3==1.34.0  # AWS
google-cloud-billing==1.11.0  # GCP
azure-mgmt-costmanagement==4.0.0  # Azure

# LLM (will be used in 1.8)
openai==1.6.0
anthropic==0.8.0

# Utilities
python-dotenv==1.0.0
httpx==0.25.2
tenacity==8.2.3

# Testing
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
pytest-mock==3.12.0
```

4. Dockerfile

dockerfile

```dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    postgresql-client \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .

# Expose port
EXPOSE 8001

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
    CMD curl -f http://localhost:8001/api/v1/health || exit 1

# Run application
CMD ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "8001"]
```

✅ Verification Checklist

Since this was completed in P-03, verify it's ready for Phase 1:

1. File Structure

bash

```bash
# Verify all files exist
ls -la services/cost-agent/src/
ls -la services/cost-agent/tests/

# Expected files:
# src/main.py
# src/config.py
# src/api/health.py
# src/api/analyze.py (from P-04)
# src/metrics.py
# tests/test_health.py
```

2. Service Running

bash

```bash
# Start the service
cd services/cost-agent
python -m src.main

# Should see:
# INFO:     Started server process
# INFO:     Waiting for application startup.
# INFO:     Application startup complete.
# INFO:     Uvicorn running on http://0.0.0.0:8001
```

3. API Endpoints

bash

```
# Test health endpoint
curl http://localhost:8001/api/v1/health
# Expected: {"status":"healthy","timestamp":"...","version":"1.0.0","database":{...}}

# Test readiness
curl http://localhost:8001/api/v1/ready
# Expected: {"status":"ready"}

# Test liveness
curl http://localhost:8001/api/v1/live
# Expected: {"status":"alive"}

# Test metrics
curl http://localhost:8001/metrics
# Expected: Prometheus metrics output
```

4. Database Connections

bash

```bash
# Check PostgreSQL
curl http://localhost:8001/api/v1/health | jq '.database.postgres'
# Expected: "healthy"

# Check ClickHouse
curl http://localhost:8001/api/v1/health | jq '.database.clickhouse'
# Expected: "healthy"

# Check Qdrant
curl http://localhost:8001/api/v1/health | jq '.database.qdrant'
# Expected: "healthy"

# Check Redis
curl http://localhost:8001/api/v1/health | jq '.database.redis'
# Expected: "healthy"
```

5. Prometheus Metrics

bash

```bash
# Check cost agent metrics
curl http://localhost:8001/metrics | grep cost_savings_total
# Expected: cost_savings_total{...} 0.0

# Check base metrics
curl http://localhost:8001/metrics | grep requests_total
# Expected: requests_total{...} N
```

🔗 Integration Points

The Cost Agent Skeleton integrates with:

1. Orchestrator (0.6-0.8)

- Registers with orchestrator on startup
- Receives optimization tasks
- Reports status and results

2. Databases (0.2-0.4)

- **PostgreSQL**: Stores cost recommendations, executions
- **ClickHouse**: Stores time-series cost metrics
- **Qdrant**: Stores learned optimization patterns
- **Redis**: Caches analysis results

3. Shared Utilities (0.10)

- Uses database connection utilities
- Uses logging utilities
- Uses configuration management
- Uses retry decorators

4. Monitoring (0.11)

- Exposes Prometheus metrics
- Scraped by Prometheus server
- Visible in Grafana Cost Agent dashboard

📊 Current Capabilities

The skeleton already supports:

✅ **Health Monitoring**

- Comprehensive health checks
- Database connectivity verification
- Kubernetes readiness/liveness probes

✅ **Basic Workflow** (from P-04)

- LangGraph state management
- 3-node workflow (Analyze → Recommend → Summarize)
- POST /analyze endpoint

✅ **Spot Migration** (from P-05)

- Complete spot migration workflow
- Multi-agent coordination
- Gradual rollout (10% → 50% → 100%)
- Quality monitoring

✅ **Metrics Collection**

- Cost savings tracking
- Recommendation counting
- Execution duration tracking
- Custom Cost Agent metrics

✅ **Database Integration**

- PostgreSQL for persistent data
- ClickHouse for time-series metrics
- Qdrant for vector embeddings
- Redis for caching

🚀 What's Missing (To Be Added in Phase 1)

The following components will be added in subsequent prompts:

Week 2 (Prompts 1.2-1.7)

⬜ **1.2**: AWS Cost Collector (boto3 + Cost Explorer) ⬜ **1.3**: GCP Cost Collector (google-cloud-billing) ⬜ **1.4**: Azure Cost Collector (azure-mgmt-costmanagement) ⬜ **1.6b**: Reserved Instance Workflow ⬜ **1.6c**: Right-Sizing Workflow ⬜ **1.7**: Analysis Engine (idle detection, anomalies)

Week 3 (Prompts 1.8-1.15)

⬜ **1.8**: LLM Integration (OpenAI/Anthropic) ⬜ **1.9**: Recommendation Engine (prioritization, scoring) ⬜ **1.10**: Execution Engine (safe execution + rollback) ⬜ **1.11**: Learning Loop (Qdrant storage, outcome tracking) ⬜ **1.12**: API

Endpoints (full REST API suite) ⬜ **1.13**: Unit Tests (80%+ coverage) ⬜ **1.14**: Integration Tests (E2E workflows) ⬜ **1.14b**: Performance Tests (load testing) ⬜ **1.15**: Documentation (complete API docs)

🎯 Success Criteria

The skeleton is considered complete and ready if:

✅ Service starts without errors ✅ All health endpoints return 200 OK ✅ All 4 databases report "healthy" ✅ Metrics endpoint exposes Prometheus metrics ✅ Can handle basic HTTP requests ✅ FastAPI docs accessible at /docs ✅ Integrates with shared utilities ✅ Base workflow from P-04 works ✅ Spot workflow from P-05 works

---

📁 Files Already Created

From P-03 (Skeleton):

- [ ] `services/cost-agent/src/main.py`
- [ ] `services/cost-agent/src/config.py`
- [ ] `services/cost-agent/src/api/health.py`
- [ ] `services/cost-agent/requirements.txt`
- [ ] `services/cost-agent/Dockerfile`
- [ ] `services/cost-agent/README.md`

From P-04 (LangGraph):

- [ ] `services/cost-agent/src/workflows/state.py`
- [ ] `services/cost-agent/src/workflows/cost_optimization.py`
- [ ] `services/cost-agent/src/nodes/analyze.py`
- [ ] `services/cost-agent/src/nodes/recommend.py`
- [ ] `services/cost-agent/src/nodes/summarize.py`
- [ ] `services/cost-agent/src/api/analyze.py`

From P-05 (Spot Migration):

- [ ] `services/cost-agent/src/workflows/spot_migration.py`
- [ ] `services/cost-agent/src/nodes/spot_analyze.py`
- [ ] `services/cost-agent/src/nodes/spot_coordinate.py`
- [ ] `services/cost-agent/src/nodes/spot_execute.py`
- [ ] `services/cost-agent/src/nodes/spot_monitor.py`
- [ ] `services/cost-agent/src/api/spot_migration.py`

From 0.11 (Metrics):

- [ ] `services/cost-agent/src/metrics.py`

🔄 Next Steps

Since 1.1 is already complete, proceed to:

NEXT: PROMPT 1.2 - AWS Cost Collector

- Integrate with AWS Cost Explorer API
- Collect EC2, RDS, Lambda costs
- Analyze spending patterns
- Identify optimization opportunities

Document Version: 1.0
Status: ✅ Already Complete (from P-03)
Last Updated: October 21, 2025
Next: PHASE1-1.2 (AWS Cost Collector)